Operating Systems Security – Assignment 5

Version 1.0.0 – 2014/2015

Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands.

1 Compartmentalization with `chroot`

Each process in UNIX knows the root of the filesystem (typically denoted /). The **`chroot`** system call changes this root and the **`chroot`** utility starts a process with a different filesystem root. The **`chroot`** mechanism can be used for *compartmentalization*: A process that is running in a **`chroot`** environment can only access files that are below its filesystem root. It can also be used to test new systems. For example, one can install the development branch of a Linux distribution in a **`chroot`** environment and test it without having to reboot. One can also use it as a development environment; for example, it is possible to run a Debian Linux system in a **`chroot`** environment on an Android phone (after rooting the phone; see, for example, https://cryptojedi.org/misc/nexuss-debian.shtml). The **`chroot`** mechanism for compartmentalization can be used to add a certain level of security, but it has various limitations, which we will investigate in the following.

Prerequisites

A process running in a **`chroot`** environment needs various files (libraries etc.) accessible. The easiest (but not necessarily most secure) way to achieve this is to make a whole UNIX environment available. Debian Linux and derivatives allow to "install" the whole environment in a directory, for example, in **`/tmp/debian`**, as follows:
`debootstrap --arch amd64 jessie /tmp/debian/ http://ftp.nl.debian.org/debian/`
This is going to take a while; afterwards you can (as root) **`chroot`** into this enviroment by running
`chroot /tmp/debian`
The environment is a quite minimal UNIX environment, so you might want to install additional software, for example (inside the **`chroot`** environment):
`apt-get install gcc`
The **`chroot`** compartmentalization does, by design, not prevent root to break out of the **`chroot`** "jail". The way to break out of the jail, for root, involves the following steps:

1. Create a subdirectory in the current **`chroot`** environment (**`mkdir`** standard C library function);
2. (open the current working directory using the **`open`** syscall);
3. use the **`chroot`** syscall to **`chroot`** into the subdirectory created in step 1;
4. (change the working directory back to the original working directory with the **`fchdir`** syscall);
5. perform **`chdir("..")`** syscalls to change to the actual (non-**`chroot`**) root of the filesystem.

The two steps in parantheses are only required if the **`chroot`** system call also changes the working directory to the **`chroot`** directory. Note that after step 4, the process has a working directory outside the current root directory; this is what allows the process to change the working directory further up to the actual root.

Objectives

a) Create a symbolic link from somewhere inside **`/tmp/debian/`** to somewhere outside **`/tmp/debian`**. Can you follow the symbolic link when using **`/tmp/debian`** as a **`chroot`** jail?
b) Create a hard link from somewhere inside **`/tmp/debian/`** to somewhere outside **`/tmp/debian`**. Can you follow the symbolic link when using **`/tmp/debian`** as a **`chroot`** jail?

c) Write a program that, when executed inside the **`/tmp/debian chroot`** jail with root rights, reads the file **`/tmp/outside`**, which is *outside* the **`chroot`** jail and outputs its contents. Submit the source code of the program.
   Note: The program will first have to escape the **`chroot`** jail using the above sequence.
d) Does the program also function without root permissions? Explain why or why not. **Note:** You can use **`chroot --userspec USERNAME`** to try this.

2 Trust models

In this exercise we consider a reference monitor which uses mandatory access control (MAC) to implement the Bell-LaPadula and the Biba model. The Bell-LaPadula model uses levels **unclassified** $\leq$ **confidential** $\leq$ **secret** $\leq$ **top secret**. The Biba model uses levels **untrusted** $\leq$ **user** $\leq$ **application** $\leq$ **system** $\leq$ **trusted**. The following objects with corresponding secrecy and trust levels are used in this exercise:

– `/home/peter/database` (confidential, user),
– `/etc/password` (confidential, trusted)
– `/etc/shadow` (top secret, trusted)
– `/usr/bin/someprog` (unclassified, application)
– `/usr/lib/somelib.so` (unclassified, system)
– Network socket to 203.0.113.42, port 80 (unclassified, untrusted)

Objectives

a) For each of the following steps determine whether the reference monitor will allow the action. If not, explain why not (if there are multiple reasons, state all).
   i User peter logs in with clearance (secret, application) and tries to run `/usr/bin/someprog`.
   ii The process dynamically loads (reads) `/usr/lib/somelib.so`.
   iii The process reads `/home/peter/database`.
   iv The process writes data to the network socket.
   v The process reads `/etc/password`.
   vi The process writes `/etc/shadow`.
b) The process from part a) now creates a new file `/home/peter/out`. What are the permitted pairs of trust and secrecy level for this output file?

3 Covert channels (again)

Virtualization (as with vmware, virtualbox, xen or other solutions) significantly reduces covert channels, however it does not fully eliminate covert channels (and side channels).

Objectives

a) List covert channels that are eliminated by virtualization.
b) List covert channels that are not or only partially eliminated by virtualization.
c) Write a program that communicates through a covert channel from one VMWare virtual machine to another VMWare virtual machine.
   Note: The program does not have to have a large communication bandwidth. It is sufficient if the sender sends one bit and the receiver receives this one bit with high probability.